Parveen Khurana

124 Followers About Following

Ensemble Methods and the Dropout Technique

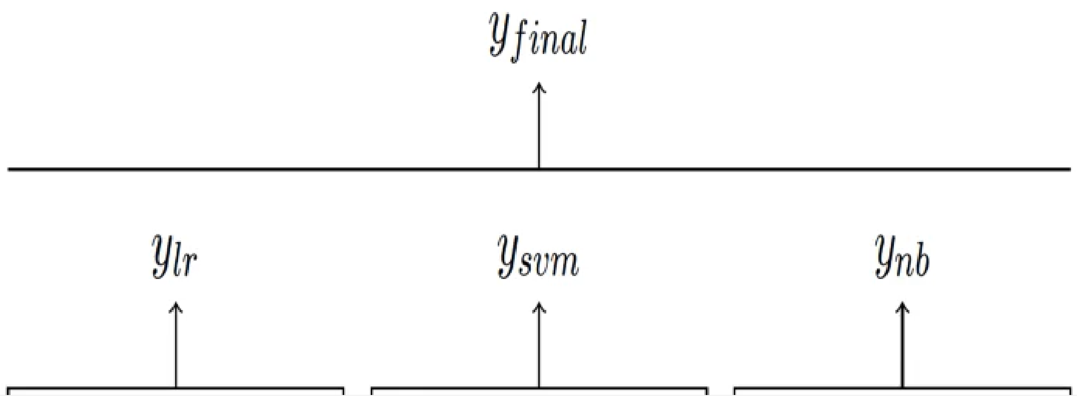
P Parveen Khurana Feb 24, 2020 · 11 min read

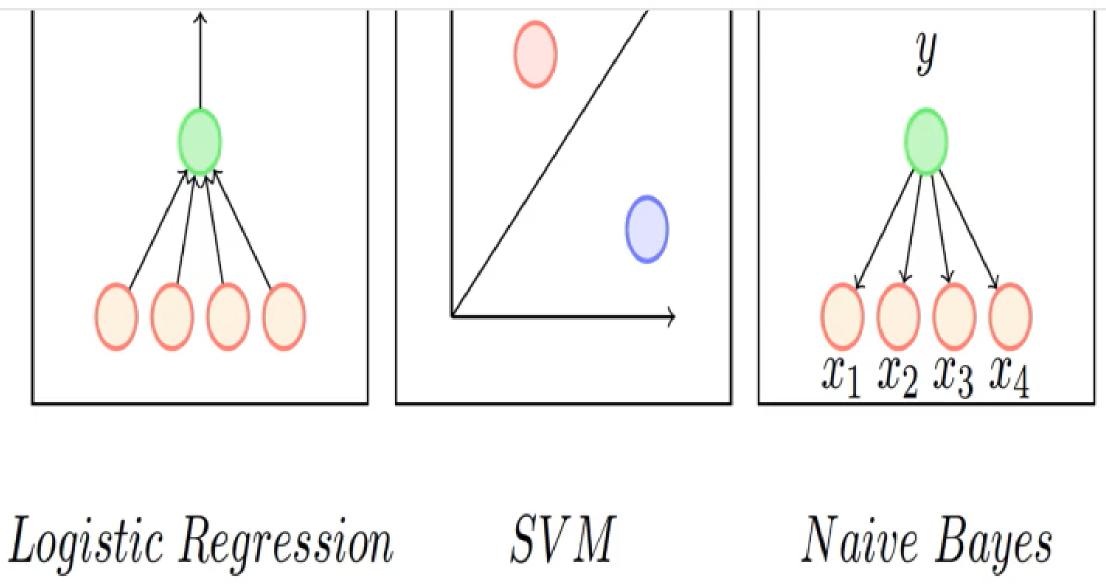
This article covers the content discussed in the Batch Normalization and Dropout module of the Deep Learning course and all the images are taken from the same module.

Ensemble Methods:

Let's say we are given some data(**'X'**) and the true labels(**'y'**), we can use any of the ML/DL algorithms to approximate the relationship between the input and the output, for example, we can approximate the relationship using Logistic Regression or say using an SVM or Naive Bayes algorithm.

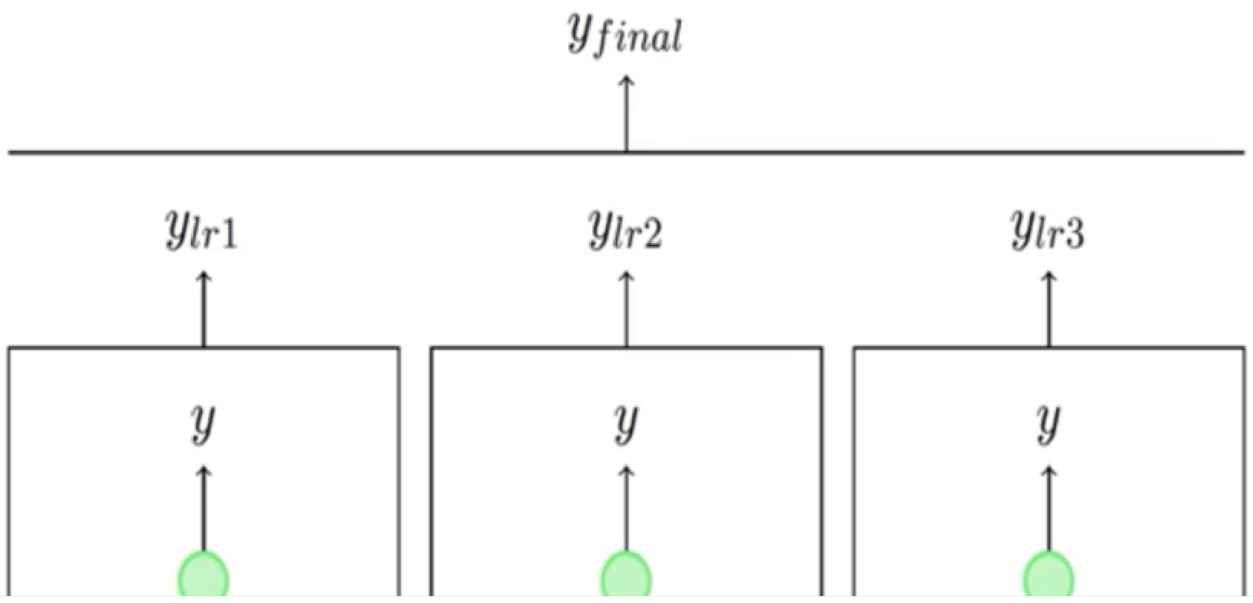
Now to make sense, instead of relying on the output of one these models, we could rely on the output of all the 3 models, then give the final output based on some census or voting or some aggregation on the output given by 3 models for example if all the models are giving us some probability value, then we could take the average of all 3 values so that we don't make an error based on the output of any one model.

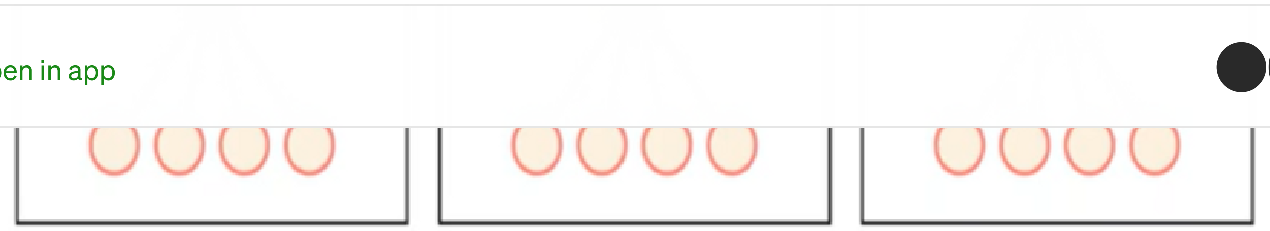




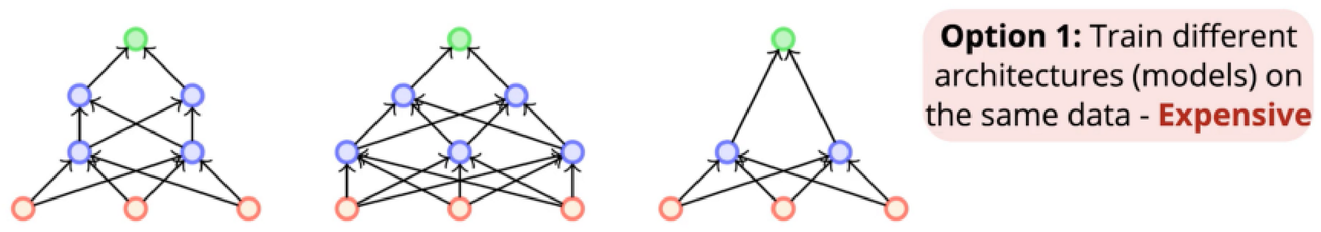
So, this is the idea behind the Ensemble Methods, we train multiple models to fit the same data and then at test time we take the aggregation or some voting of output from all these methods, and this aggregation could be the simple average or it could be some weighted average.

Now to take the idea forward, it might be the case that all the models/functions(to be used when ensembling) are the same but either we train them on different subsets of the data or we could train them on different subsets of the features say we have trained model 1 using some of the features, model 2 using some other combinations of the features or we could have the trained the model using different hyper-parameters. So, we could get different models from the same family of functions by using any of the ways.

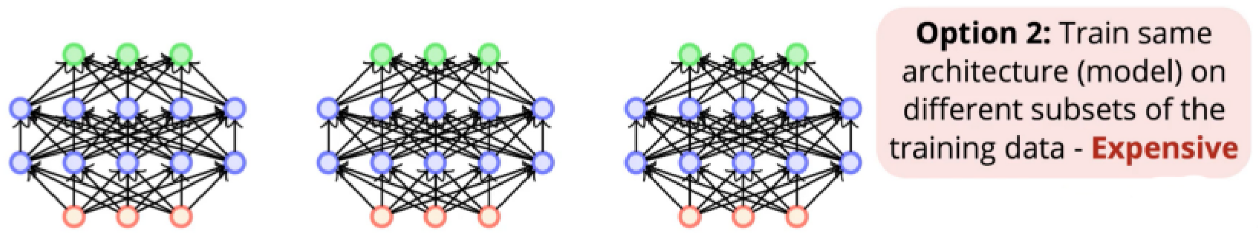




Now in our case, we want to have an Ensemble of neural networks and we have two options:


Option 1: Train different architectures (models) on the same data - **Expensive**

This option could be expensive as training just one neural network requires a lot of computations and in this option, we are training different neural networks having a different number of intermediate layers, neurons and so on and then we are ensembling them, so this option is going to be computationally expensive.


Option 2: Train same architecture (model) on different subsets of the training data - **Expensive**

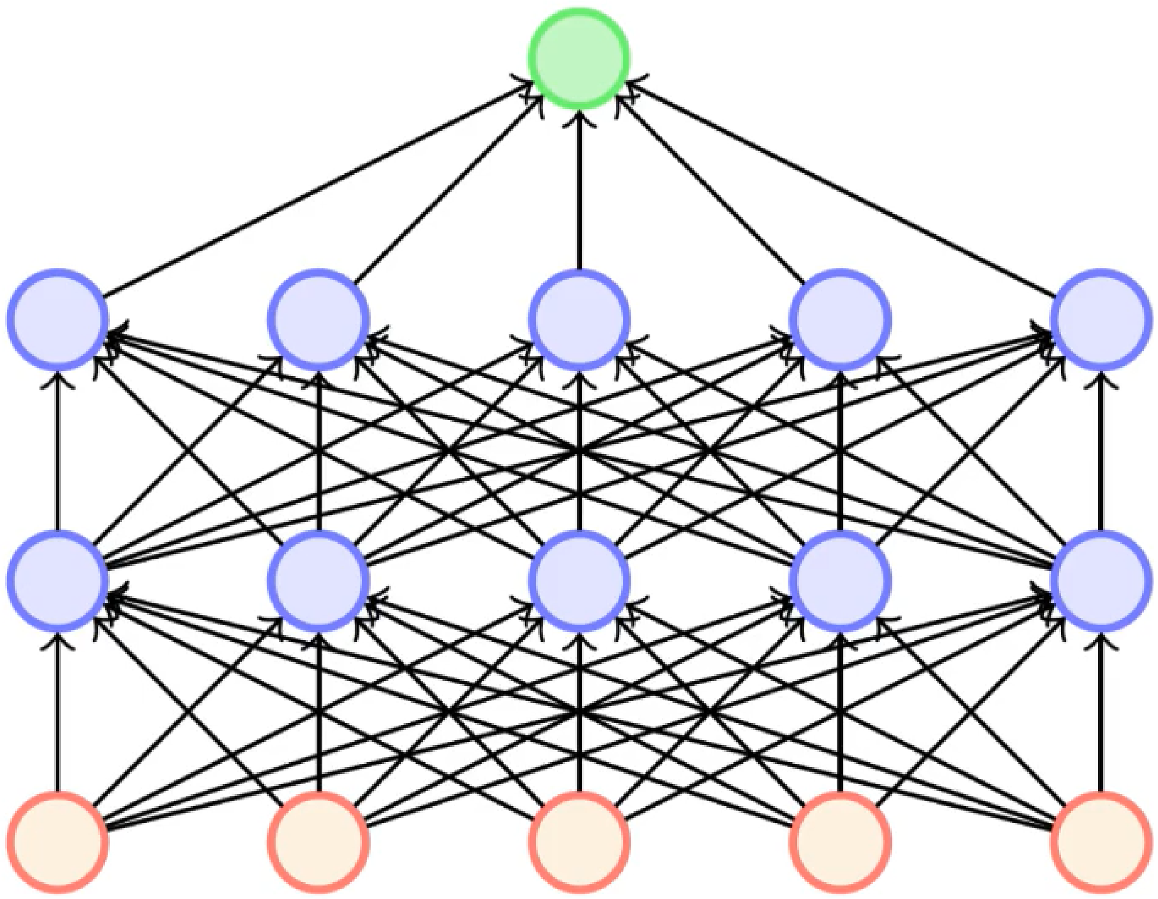
Option 2 is we have the same model architecture but we could either train them on different subsets of the training data or different combinations of the features. So, in this option, we have the same family of functions but we are training them on different subsets of data or on different features so the value of the parameters that we will get would be different for each of the models. This option is also expensive as we are training 5–6 different neural networks model and then taking an ensemble of them.

We want to take the benefit of the Ensemble technique for neural networks but both of the options are expensive. To solve this problem, we look at the concept of Dropout.

The idea of Dropout:

- Acts as a regularizer by introducing noise
- Prevents co-adaptation

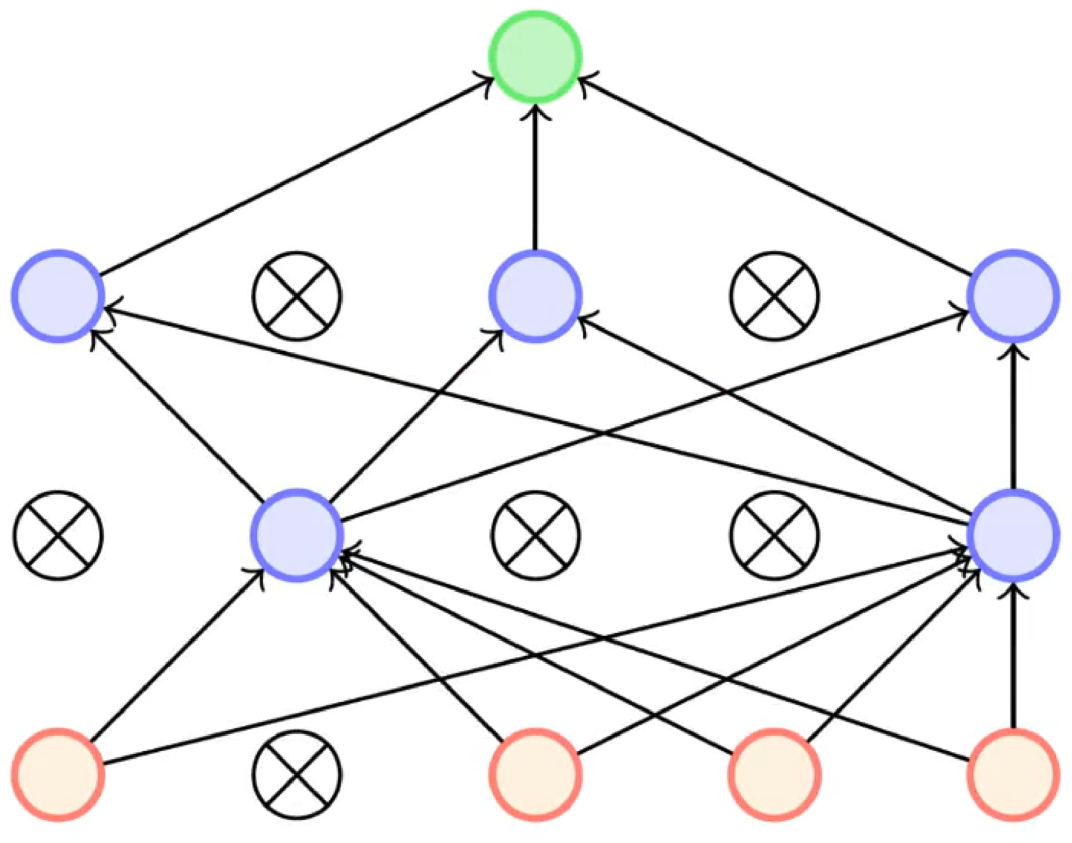
The first question that we ask here is that given a neural network, can we create multiple neural networks? Let’s say the below network is given to us



Original network

We are interested in creating an ensemble of neural networks using the original architecture given to us. Let’s say we have some inputs, 4 nodes(neurons) are there in the network then we have a final output neuron/node.

As we want to create different neural networks from the given architecture, **we can think of each node as to whether to keep it in the network(say value 1) or to drop it(say value 0)**. So, for each of the nodes, we could either have the value 0 or 1. As architectures(different neural networks) possible where say in one architecture all the nodes are present, in another one, first three nodes are present and the last one is missing and so on for the remaining 14 architectures.



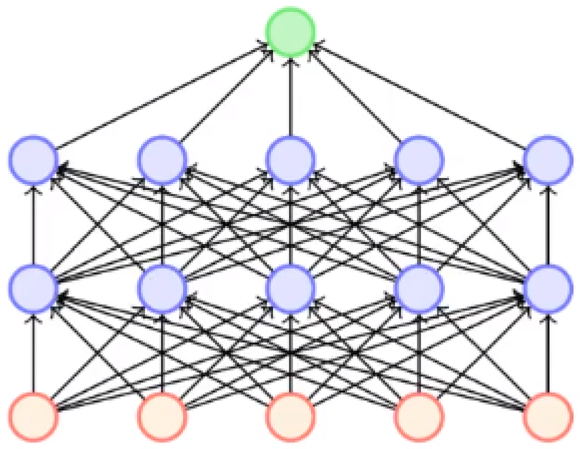
Network with some nodes dropped out

In general, if we have '**n**' nodes/neurons in the network, then a total of '**2^n**'(2 raised to power '**n**') architectures possible and each of these corresponds to some of the nodes being present and some being absent. This the idea behind the dropout and is clear from the above image is that some of the nodes(crossed nodes in the above image) get dropped from the network. This also takes care of the Ensemble problem, we have Ensemble of different neural networks and all of those are created from one single network/architecture dropping some nodes from it.

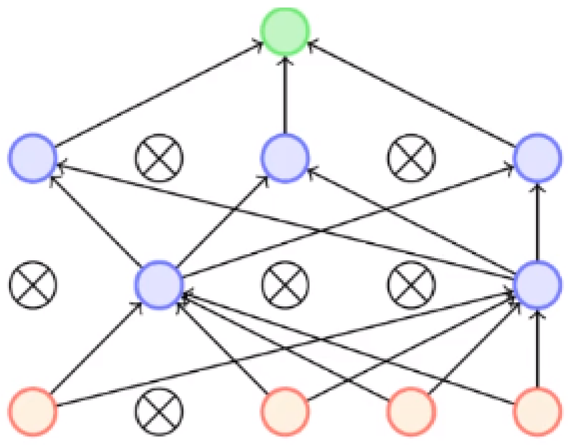
Now say the value of '**n**' is **100**, then we have a total of $\mathbf{2^{100}}$ networks possible, so the question is how do we train these many networks. Dropout solves this problem using two tricks:

Trick :

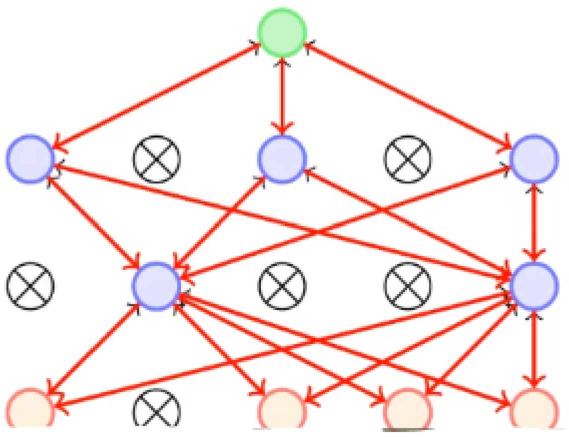
Let's see this in Algorithmic form, say we have the below original architecture:



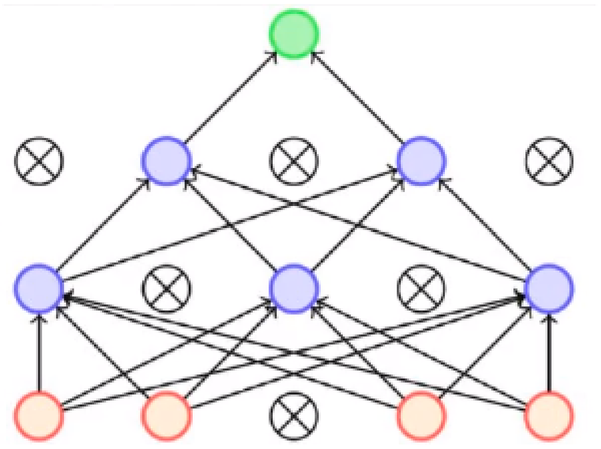
We initialize the parameters randomly, we are given some data(**'X')**, **we iterate over the data**(we typically do this in batches), and **sample the current mini-batch** and we are going to train based on this mini-batch, now **we create a neural network which is dropped out version of the initial neural network**(the way we do this is that we sample a random number between 0 to 1 for every node and if this number is greater than some threshold say 0.5 then we retain that node else we drop it and if a node is dropped then there are no incoming/outgoing weights to/from this node), say the dropped out network looks like the below:



network architectures from the original one). Once we have this dropped out network architecture, we proceed with the regular stuff i.e to pass the inputs through the network, computing an output based on which then we calculate the loss value and use it for backpropagation to update the weights(we only update those weights which are used to compute the output value, some weights are dropped from the network we don't update them)

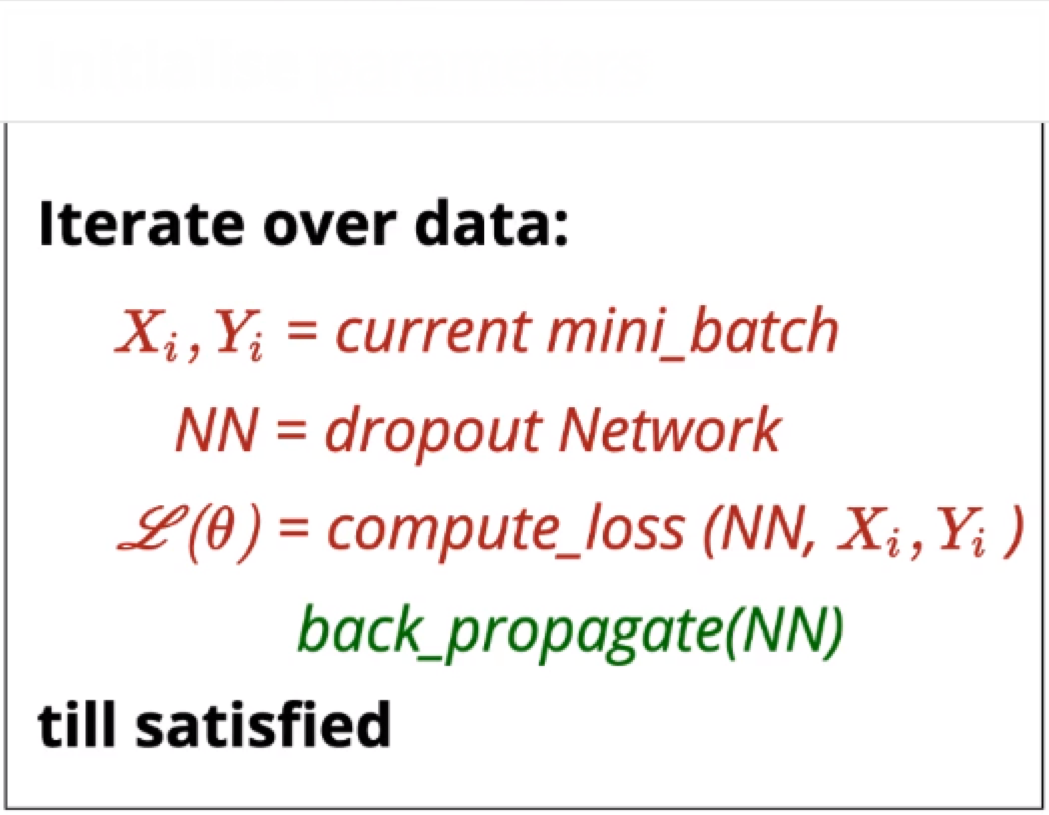


So, this is what we do for the first mini-batch of the data. Then we iterate over the dataset and pick the second mini-batch, we again apply the dropout on the original network and we get a different network which is different from both the original network and the dropped out network used with the first mini-batch:



Again, we do the same thing i.e pass the input through the network, compute the output, loss value, backpropagation and update the relevant weights. So, the algorithm looks like:

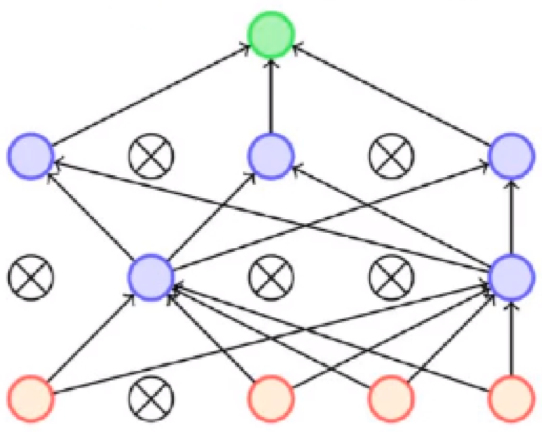
Open in app

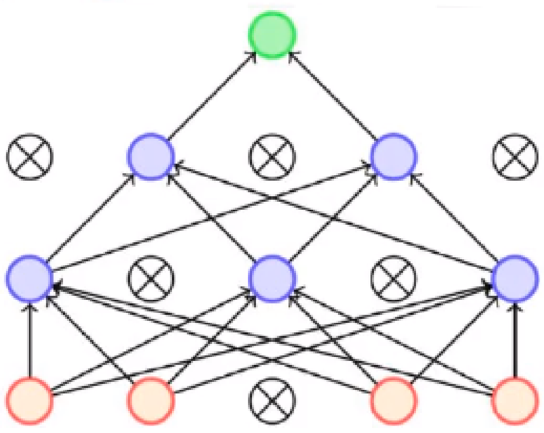


```
Iterate over data:
    $X_i, Y_i$ = current mini_batch
    NN = dropout Network
    $\mathscr{L}(\theta)$ = compute_loss (NN, $X_i, Y_i$ )
    back_propagate(NN)
till satisfied
```

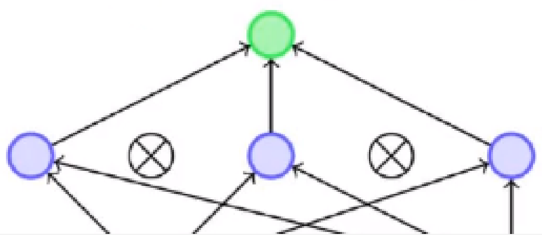
Let’s see how weights sharing helps us:

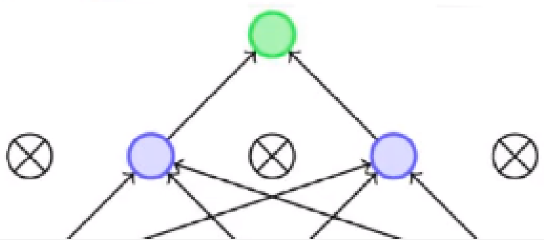
In the above case, we have seen two different dropped out architectures from the original network

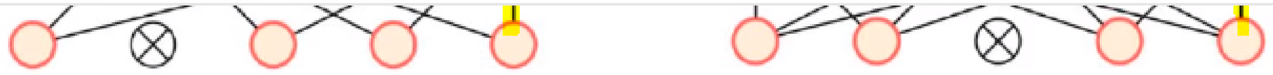




There are some weights which are present in both the networks as depicted below







Let's say we call this highlighted weights as '**ao**' when we have the first dropped out architecture, this weight was present and we update it as per the optimization algorithm update rule, say we call it as '**a1**' after the update. Now when we sample the next batch and have the new dropped out architecture, this weight is present and here we are going to use the value '**a1**' to use in the update rule and not the very initial value which was '**ao**' that means whatever the updates were done in the previous iteration are carried over to the next iteration.

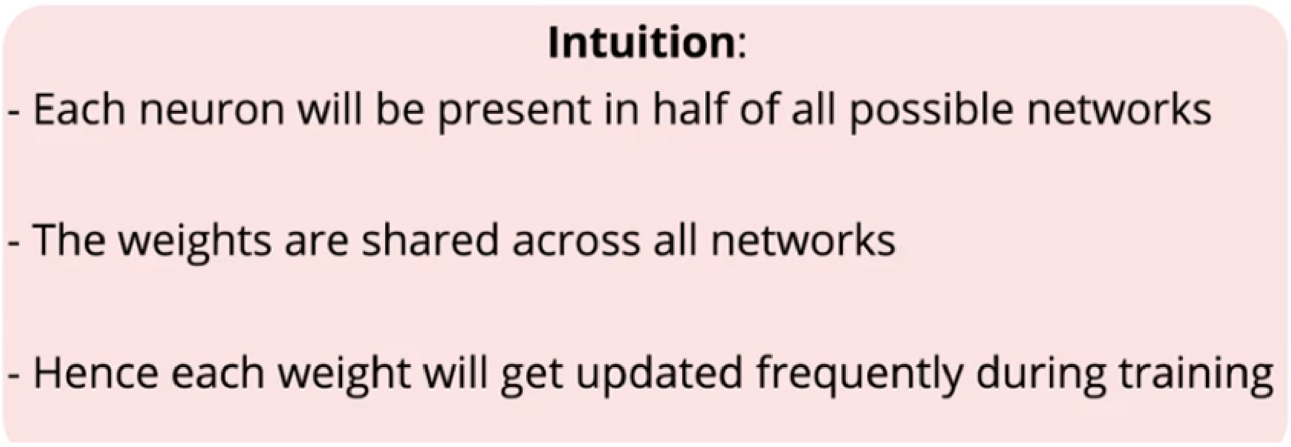
So, even though we are creating a new network architecture for each mini-batch that we sample, we are using the same weights across all the networks or the weights are shared and the same weights are continuously getting updated across these different neural networks.

Now, we have a large number of possible network architecture say if we have 100 nodes then the possible number of networks are $\mathbf{2^{100}}$ and even if we have a million updates of the parameters, still there would be some networks which we are not able to sample from the possible set of neural networks. So, that means there would be some networks that never get sampled and some networks that get sampled frequently. So, the question is **how do we ensure that the training of the networks is efficient?**

Let's answer this question in two steps:

1. We could either have '**M**' different networks each training for say '**k**' number of epochs, so, in this case, a total of '**Mk**' steps of training which we do not want. What we are doing is that we are starting with one network, doing '**k**' steps of training and for each step, we are sampling one of the variants of the original network that we have. So, we have a large number of possible networks but each network is getting trained a few times.
2. Let's take a particular weight say '**w32**' in the network. Our job is to ensure that this weight '**w32**' gets updated enough number of times(out of the '**k**' steps for which we are training/updating the weights).

least half the possible networks; that means for every batch the possibility of a node being present is 50% or 0.5 so what this means is that every weight could actually get updated at least 50% of the times i.e if we train the network for ‘**2k**’ steps then all weights would get a chance to get updated at least ‘**k**’ times. So, that’s why we training for fewer steps but still, all the weights would get updated and that’s what we actually want.


Intuition:

- Each neuron will be present in half of all possible networks

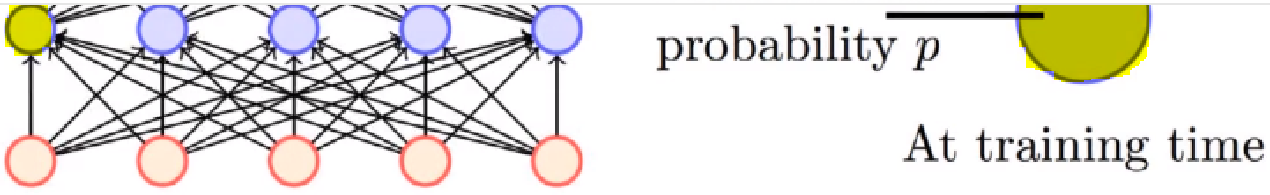
- The weights are shared across all networks

- Hence each weight will get updated frequently during training

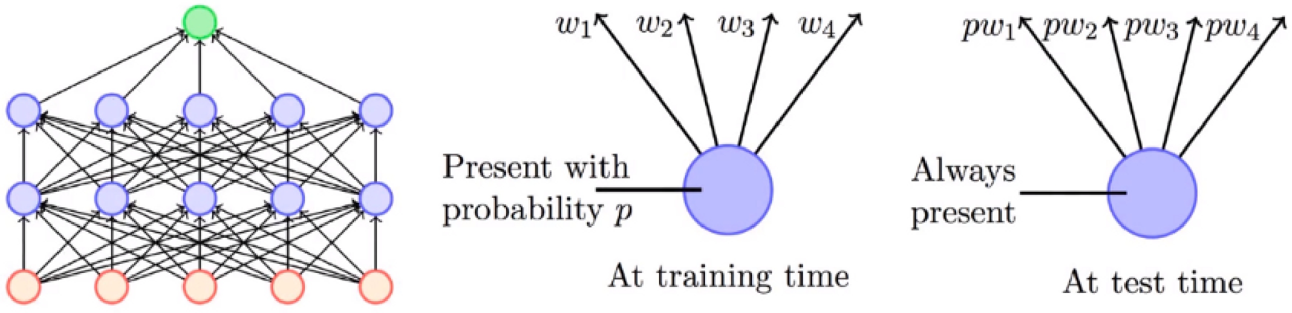
As discussed earlier in this article, at test time, we pass the inputs through multiple networks, compute the output and then take some kind of average over these output values from different networks and consider that average as the final output.

Now we have ‘**2^n**’ networks possible and if we pass the input through all these networks, then again we would run into the computational problem. So, how do we use the network at test time is the question here that we want to answer?

If we look at any node(say the one in yellow in the below image) at training time, this node is present with the probability ‘**p**’(say we want to keep a node with probability 0.8 then what we do is that we sample a random number between 0 and 1 and if this number is greater than 0.2 than we treat it as 1 i.e we keep the node else we consider it as 0 and drop that node), since any node is present with probability ‘**p**’ at the training time that means the weights associated with this node gets updated only ‘**p**’ fraction of times



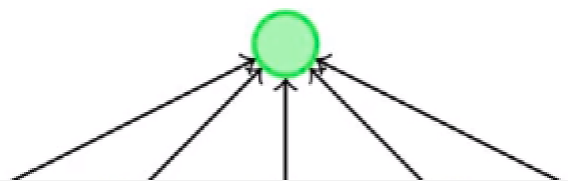
What we do at test time is that is we use the large original network architecture(we don't use an ensemble) and at test time whatever this node(the one with probability '**p**') outputs, since it is only '**p**'% reliable, so we are going to scale its output by '**p**'. So, now instead of passing the inference through all the possible neural networks, we are passing the input through one network and computing the output with the catch that instead of using each node as it is, we are simulating the same thing that these nodes are present '**p**'% of the times by scaling the output appropriately.

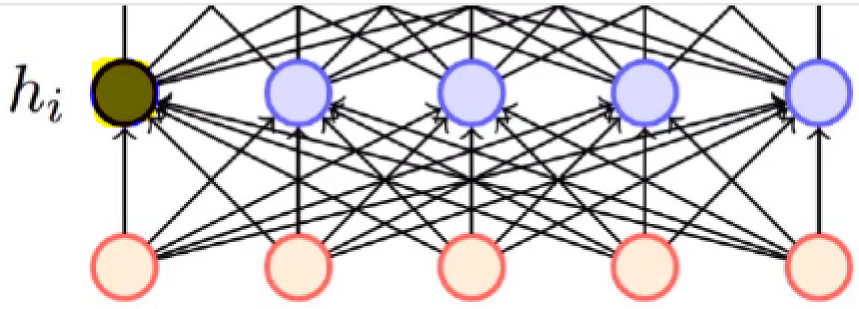


Intuition:
Scale the output of each neuron by *p*

Let's see how dropout acts as a regularizer:

Say we are building a human face classifier using the below neural network and node in yellow in the below image fires whenever it sees a nose in the picture and some other node fires if eyes are there in the picture:





All the neurons in the network could do something known as co-adaption, what it means is that say the neuron in yellow in the above image becomes good at recognizing eyes in the pictures and because of that it contributes to the next layer and predict the final output correctly, and what happens is that all the other neurons becomes kind of lousy that this neuron(in yellow) is detecting the eyes correctly and predicting the output correctly so we don't need to do much and they are in a way co-adapting or depending on this neuron. Now, what happens when we add dropout is that there would be instances when this particular node(which detects eyes) is not present at all and all of the other neurons needs to rise up to the occasion and they should still be able to learn enough so that the final output is correct.

So, some of the other neurons must also get good at detecting eyes in the input, so dropout in a way makes the network more robust, it prevents this co-adaption(neurons adapt to each other in a way that you do this job I will do some other job), this kind of thing dropout prevents as suddenly in the next batch, some neurons would disappear and we can no longer rely on them. This is similar to the idea of introducing noise in the input.